Electronic Notes in Theoretical Computer Science 82 No. 5 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html

Design Aspects and GRS-based AOD

The GREAT transformation framework

Alexander Christoph[1]

SWT
Forschungszentrum Informatik[2] *(FZI)*
Karlsruhe, Germany

1. Email: christo@fzi.de
2. URL: http://www.fzi.de

Abstract

In analogy to Aspect Oriented Programming (AOP), Aspect Oriented Design (AOD) is a means of modelling different concerns in software designs independently from each other and from the design itself. Different approaches to AOD formalize design aspects as new design entities. Unfortunately, these approaches are difficult to use or become unhandy with existing or large designs. This paper presents the rule-based design transformation framework GREAT that can be used to describe aspects in a formal way and generate weaving code from aspect descriptions.

1 Introduction

The separation of concerns on the level of source code has been studied in a number of papers [10], [12]. The key issue of aspect oriented programming (AOP, [9]) is to realize different concepts separately from the implementation of the program. These concepts are later woven into the program, either prior to or during its execution. AOP aspects are defined as patterns over the program call graph, i.e. the execution view of the software. This facilitates the developer to declare additional behaviour before, during or after a method has been called.
In [5] the authors explain that modelling different concerns separately during the design process can be useful, too. Though an aspect model for software designs must be based on the design view of the software. Different approaches exist that introduce aspects as new entities in the structural specification of a program. Unfortunately, most of them require the developer to clearly specify, which parts of the design are affected by aspects by connecting aspects and their point-cuts (variation points) via special relationships. This contradicts the AOP idea, in which the developer uses patterns to declare at which call sites modifications are required.
This paper introduces the rule-based transformation system GREAT that uses graph rewriting for specifying software design transformations. GREAT facilitates the developer to (1) describe design aspects separately from the software design, (2) use patterns to describe points of interest and (3) generate and execute weaving code that can modify arbitrary software designs.

2 Related work

In [13] the authors describe the need for separating designs and design aspects. They argue that this separation simplifies reuse of aspects and enforces the independent modelling of concerns and designs. In contrast, many of the existing approaches to AOD encapsulate design aspects as new entities in software designs. Furthermore, they require a fixed relationship (e.g. "cross-cuts") between aspects and their point-cuts ([8], [14]). This can become difficult or even impossible for large designs, especially when multiple aspects have to be woven to one class.
UMLAUT [7] is an appealing approach to AOD, since it defines a transformation framework based on the iteration of lists. The user can specify transformation rules in terms of filter expressions that can be nested to match complex structures. These filters are evaluated by the UMLAUT framework. UMLAUT mainly focuses on the transformation of software models for simulation purposes.

3 Graph-based AOD

In [3] the authors describe their approach to weave aspects by specifying them as graph rewrite rules. The main contribution of their work is the (re-)definition of aspect related terms in respect of graph rewrite systems. Their definitions are reformulated here in terms of AOD.

3.1 Definitions

Host graph A host graph is the graph to be rewritten by the graph-rewrite-systems. In AOD this refers to the software design, more precisely its static structure (e.g. UML class diagram).

Context-free pattern A context-free pattern is a finite connected graph.

Context-sensitive pattern A context-sensitive pattern is a graph of at least two disconnected subgraphs.

Redex A redex is a subgraph of the host graph injectively homomorphic to a pattern. In AOD this is an instance of a design join point.

Graph rewrite rule A graph rewrite rule r = (L, R) consists of a left-hand side pattern L and right-hand side graph R. L is matched in the host graph, i.e. is redex is found in the host graph.

Direct derivation A rewriting step which replaces a redex with parts specified in the right-hand side of a rule.

Derivation A sequence of direct derivations.

The previous definitions enable the definition of GRS-based AOD.

Join point A join point is a redex in the software design graph.

Aspect pattern An aspect pattern is the graph pattern in a left-hand side of a graph rewrite rule.

Aspect An aspect contains an aspect pattern and a right-hand side graph together with activities which modify the identified join points.

3.2 The join point model

There are different types of join points for AOD.

Matchable join point A matchable join point can be directly matched by an aspect pattern. The pattern matching can rely on structural information as given in the different relations. In addition, the pattern can access node (class) attributes, such as class names, declaration of features, etc.

All design elements are matchable join points. Examples are class declarations, feature declarations, etc. Also relationships are matchable: associations between two classes, inheritance relationships with a certain stereotype, etc.

Computable join point Computable join points cannot be matched immediately. First, a preprocessing phase is required that incorporates implicit knowledge to explicit knowledge. An example are dependency relationships that are used to build a reflexive closure over all inheritance relationships in a design. Using the closure, all parents of a class can easily be matched.

Declared join point Join points can be declared explicitly in the software design. UML offers extension mechanisms (i.e. stereotypes and tagged values) to provide design elements with textual extensions. These informations can be part of an aspect pattern.

3.3 Types of Rewrite Systems

In order to address the critical issues of termination, indeterminism and confluence, Assmann identified different types of rewrite systems. According to this classification, there exist three different types of aspects for GRS-based AOD. The different rule types are illustrated with examples written in pseudo-code.

Edge-additive rules (EARS rules) Edge additive rules are a simple form of a rewrite rule. They just add relations to the redex. These rewrite systems are always terminating and lead to a unique normal-form. Typically, edge-additive rules are used to analyse designs in a preprocessing phase. An example is the construction of a transitive closure over inheritance relationships for inferring type relations, shown in Figure 1.

```
EARS transitiveClosure(root:Root) :-
  let M1 be a Metaclass,
  let E1 be an Extension from M1 to M2
==>
  connect M1 and M2 with a Dependency edge stereotyped "transitive closure"

  let M1 be a Metaclass,
  let D1 be a Dependency from M1 to T stereotyped "transitive closure",
  let E1 be an Extension from T to M2
==>
  connect M1 and M2 with a Dependency edge stereotyped "transitive closure"
```

Figure 1: EARS aspect

Additive rules Additive rules add information to the redex. They do not modify the join points. In consequence, redexes are never destroyed. [2] shows, that these rewrite systems yield a unique normal form, if they terminate.
Termination of such rewrite systems has also been formalized. As long as such a rewrite system completes the redex of the host graph and does not add nodes to it, it terminates. In case of soft-

ware designs, it even can add nodes, as long as nodes have a unique name (key attribute) and the name allocation function in the aspect delivers a finite set of identifiers.

```
GRS featureSet(root:Root) :-
  //...
  let MC be a metaclass stereotyped "fixed",
  let M be a method in MC
==>
  let FC be a new metaclass(nameOf(M)) stereotyped "feature class",
  connect FC and MC with a Dependency edge stereotyped "occurs in"
```

Figure 2: Creating new metaclasses

Figure 2 shows a part of a rule that creates a metaclass for every method in metaclasses that are stereotyped “fixed”. This rule terminates, since the set of metaclasses matching the aspect pattern and their methods is finite.

Node modifying rules General rewrite systems are able to modify the redex in any way. These rules cannot be proved to terminate, nor do they lead to a unique normal form. Also, non-commutative derivations may exist. These rules lead to indeterministic weavers. Using such rules requires the prove of semantically equality of the different derivations.

4 GREAT: An AOD transformation framework

The GREAT transformation framework [4] is an implementation of the previous ideas. It provides methods for reading and storing software designs described in UML and supports the execution of generated GRS-based AOD weavers. AOD weavers are generated by the OPTIMIX graph rewrite system [1].

4.1 Generating AOD weavers from high-level aspect descriptions

GRS based weavers are rather complicate to implement, since pattern search and graph-modifications are difficult algorithms. The use of Optimix allows for the generation of efficient graph-based weavers. Optimix generates the code of a pattern search and modification algorithm out of a graph metamodel and a rule, that describes a graph pattern with a set of predicates over the graph nodes and edges. Graph modifications are defined as (body-)predicates, that must hold after the modifications took place. Additional activities can be implemented by calling helper methods from the supporting GREAT framework.
The generated code searches subgraphs in the given design-graph that match the given set of predicates. It then applies modifications to the redex that guarantee that the given body predicates match.

4.2 Transformation framework

Generated weavers are executed in the GREAT transformation framework. The framework is responsible for loading software designs, building the design graph, executing a set of weavers and storing the modified design. Software designs are loaded and stored using the XMI [6] format, enabling the integration with existing CASE tools. Figure 3 shows the structure of the

framework. Generated aspect weavers (A1...A4 in the figure) are executed, until the design stabilizes. The transformation interface provides methods for modifying the software design.

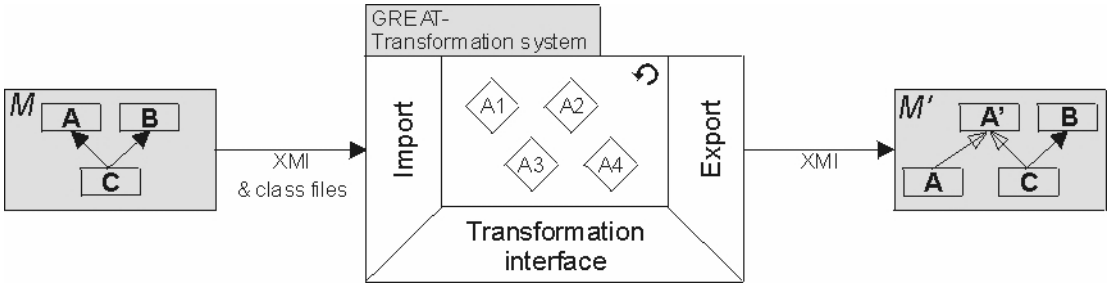

Figure 3: AOD framework GREAT

5 Evaluation

First experiments have shown the power and utility of the approach. One example is the implementation of the IHI algorithm [11] for inferring an optimal inheritance hierarchy. Another example shows, how complex class hierarchies can be sliced, using the visitor design pattern. A case-study in cooperation with a software company demonstrates, how GREAT can be used to realize the shift of a commercial software to another middleware platform.

6 Future work

Further research focuses on the dynamic parts of aspect descriptions: can dynamic specifications (collaboration and sequence diagrams) improve pattern matching for AOD? Can aspects use dynamic specifications to describe the desired behaviour of a weaving result? In some cases, the structure of join points is not sufficient to identify unique instances. Can user interaction be helpful in such cases? How can designs be enhanced to supply additional informations?

References

[1] Uwe Assmann. *Generierung von Programmoptimierungen mit Graphersetzungssystemen*. PhD thesis, Universität Karlsruhe, Fakultät für Informatik, 1996.

[2] Uwe Assmann. Graph Rewrite Systems for Program Optimization. In *ACM Transactions on programming Languages and Systems (TOPLAS)*, volume 22, 4. ACM Press, New York, NY, USA, 2000.

[3] Uwe Aßmann and Andreas Ludwig. Aspect weaving by graph rewriting. In U. W. Eisenecker and K. Czarnecki, editors, *Generative Component-based Software Engineering (GCSE)*, October 1999.

[4] Alexander Christoph. Graph Rewrite Systems for Software Design Transformations. In *Tagungsband der Hauptkonferenz NET.ObjectDays 2002*, pages 87–96, October 2002.

[5] Tzilla Elrad, Omar Aldawud, and Atef Bader. Aspect-oriented modeling: Bridging the gap between implementation and design. In *ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, October 2002.

[6] Object Management Group. OMG XML Metadata Interchange (XMI) Specification, 2000.

[7] Wai-Ming Ho, F. Pennaneac'h, and N. Plouzeau. UMLAUT: a framework for weaving UML-based aspect-oriented designs. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33), Mont-Saint-Michel, France*. IEEE Computer Society, 2000.

[8] Mohamed Mancona Kandé, Jörg Kienzle, and Alfred Strohmeier. From AOP to UML—A bottom-up approach. In *Workshop on Aspect-Oriented Modeling with UML (AOSD-2002)*, March 2002.

[9] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es):154, 1996.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[11] Ivan Moore and Tim Clement. A Simple and Efficient Algorithm for Inferring Inheritance Hierarchies. In *Proceedings TOOLS Europe 1996*. Prentice-Hall, 1996.

[12] Johan Ovlinger, Karl Lieberherr, and David Lorenz. Aspects and modules combined. Technical Report NU-CCS-02-03, College of Computer Science, Northeastern University, Boston, MA, March 2002.

[13] Stanley M. Sutton Jr. and Peri Tarr. Aspect-oriented design needs concern modeling. In *Workshop on Identifying, Separating and Verifying Concerns in the Design (AOSD-2002)*, March 2002.

[14] Junichi Suzuki and Yoshikazu Yamamoto. Extending UML with aspects: Aspect support in the design phase. In *Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999)*, June 1999.